COG-PETS: CODE GENERATION FOR PARAMETER ESTIMATION IN TIME SERIES

CHRISTOPHER KUMAR ANAND, JACQUES CARETTE,
ANDREW THOMAS CURTIS AND DAVID MILLER

1. Introduction

We have previously shown that symbolic computation coupled with efficient code generation can significantly simplify the development process for image and signal processing applications [1]. In that work, our application was the development of visual target tracking algorithms based on continuous optimization. Simplifying the development process means two things: faster development, and higher quality results. Faster development stems from automating steps which previously involved tedious and error-prone manipulations (like differentiation) and transcription (converting matrix multiplication to loops). Higher quality results arise because the programmer now deals at the semantic level of the declarative mathematics instead of at the level of imperative code.

Incorporating some verification steps into the process of compilation leads to a further improvement in the development process. We also showed how symbolic code generation can greatly simplify the generation of derivatives of model functions (Jacobians and Hessians), as required by nonlinear optimization based on Newton's method. Furthermore, the aggressive common subexpression elimination available through Maple's code generation facilities of the resulting code list can reduce computation time by half for typical image processing task.

In this work, we take advantage of more of the structure present in the optimization problem. This lets us perform even more symbolic computations on the model before code generation, which result in significant efficiency gains. In particular, we deal with models that satisfy a recurrence relation in the time domain and a system of linear differential equations in the parameter domain. These models are very general, as well as being pervasive in a wide number of application domains. In our trials, we observed reduction factors between 120 and 500 in computation time. Working at this level also has the potential for increased software quality by ensuring that the input models do satisfy the appropriate recurrences in time and differential equations in parameter space. As the equations are generated rather than user-entered, considerably more complex models can be handled, as these models would have been prohibitively difficult to handle by hand.

Date: 18 May 2005.

In the applications we have in mind, these spectacular speedups matter a lot: in the case of relaxometry, these calculations would be required for all pixels of an image (*i.e.* millions of solves, and thus correspondingly more inner loop calls), or in the case of spectroscopy, optimization across all patients in a clinical study means gigabytes of data processed in hundreds of iterations.

Recurrence relations can be used for efficient evaluation of mathematical objects [2], but in this work, we will restrict our attention to vector-valued functions of the real line which are sampled at evenly spaced points. This case arises naturally in image and signal processing from time-series and pixelated images. Since they are commonly composed of combinations of closed-form analytic functions, the assumption that they satisfy a linear ordinary differential equation (ODE) in the parameters is more natural than restrictive.

The novelty in this work comes from the combination of a wealth of symbolic computations, namely automatic differentiation, automatic derivation of recurrences and differential equations, automatic transformations of vectors-of-sums into sums-of-vectors, use of recurrence and differential equations for the simplification of the evaluation of the model, and finally code generation.

Taken individually, these techniques measurably reduce the complexity of the resulting expressions, which result in reduced execution time. Taken together, we have measured a 120-fold reduction in execution time for real valued exponential models when compared to a 'vanilla'implementation, and a 540-fold reduction for complex valued exponential models. Although we wouldn't expect this to be the case for all applications, we certainly expect significant gains for many applications.

There are a multitude of physical processes involving decay equations which can be efficiently computed using recurrence relations, and some of them generate quantities of data which still pose a computational challenge with modern processors. Among these are Magnetic Resonance Spectroscopy (MRS) and Magnetic Resonance Relaxometry. The first is used to identify the chemical composition within living tissue, and can be used both in clinical diagnosis and biomedical research. We will explain this application in some detail below. Relaxometry measures tissue properties which depend on changes in pH and temperature, and has been proposed as a method of diagnosis and treatment monitoring, in particular, non-invasive, real-time temperature monitoring.

The rest of this paper is structured as follows: We first present the mathematical models of interest. The next two sections outline the model-based manipulations that we perform, and the code generation infrastructure necessary for generating code from the mathematical model. We then describe one (new) application of our methods, namely Magnetic Resonance Spectroscopy. The next section gives the results of our experiments, and clearly shows the cumulative benefits of our approach. We finally draw some conclusions and outline some future work.

2. Mathematical Problem

Parameter estimation from time-series data is a challenge occurring in many problem domains including, determination of rate constants in pharmaceutical drug transport, decomposing audio signals and voice recognition, and measurement of metabolite levels in Magnetic Resonance Spectroscopy and Relaxometry.

A common method of parameter estimation for time series data involves modeling signal sources, $f(x_1, x_2, \dots, x_n, t)$, (where the x_i are the model parameters and f is in general a vector-valued function) and fitting a superposition of the various sources to the measured data. Through minimization of an objective function (commonly the difference between the model and the measurements, in either a 2-norm or χ^2 sense) an optimal set of parameters may be determined:

$$F = \min_{x_1^1, x_2^1, \dots, x_n^1, \dots, x_n^s} \sum_t \left\| y(t) - \sum_{s \in \{\text{sources}\}} a_s f_s(x_1^s, x_2^s, \dots, x_n^s, t) \right\|^2 . \tag{2.1}$$

where x_j^s denotes the x_j'th parameter of peak s. With an objective function available, a choice of optimization method must be made. Assuming the objective function is twice differentiable, algorithms based on Newton's method become an attractive choice. This is the case for a vast majority of models (they in fact tend to be analytic), and as such we restrict our discussion to the simple example of minimization via Newton's method.

2.1. **Structure of Newton Solvers.**

The example solvers given in this paper are based on multivariate Newton's method. It is well-known that if our initial point is close to the solution, Newton's method will converge. If it is farther away, it may still converge, but very slowly, and numerical errors may prevent it from converging. In general, it will not converge at all. However under mild assumptions, every local minima is contained in a neighbourhood which is contained in the basin of convergence of the local minima under the Newton iteration. Depending on the structure of the problem, it may be possible to find a series of functions which approximate the objective function and which are sufficiently nice to make a staged solver – one which solves a series of increasingly difficult problems – converge efficiently in practical cases. The most straightforward method of finding such a series of objective functions is to restrict the original function to subspaces of the original domain, minimizing only a subset $\mathcal{U} \subset \mathcal{X}$ of the model parameters.

Let $J_{\mathcal{U}}$ be the Jacobian of F and $H_{\mathcal{U}}$ the Hessian of F with respect to the variables $\mathcal{U}$. The Newton iteration is defined by the recursion:

$$u_{n+1} = u_n - H_{\mathcal{U}}(u_n)^{-1} J_{\mathcal{U}}(u_n)$$

which in practice is implemented as solving

$$H_{\mathcal{U}}(u_n)(u_n - u_{n+1}) = J_{\mathcal{U}}(u_n).$$

2.2. **Efficient Implementation.** The advantages obtained by careful mathematical modeling and optimization-based parameter extraction can easily be overwhelmed by development time, and sometimes by computational costs as well. The expense comes from two sources: freedom in choosing the target pattern/model and the need to develop multi-stage solvers to achieve convergence requirements for the highly nonlinear models. The present method of generating a family of efficient Newton solvers from any target model efficiently solves this problem. By generation of Newton solvers, we mean generation of optimized Jacobian and Hessian matrices for different sets of parameters, which are used in a basic Newton method iteration. The code generator described in this paper currently generates solvers for 1 dimensional (time) models in an arbitrary number of parameters, is being generalized to 2 dimensional grid-based models, and could easily be generalized to higher dimensions.

In model fitting against signals and images, data is stored in large arrays. Calculation of the sum over elements in arrays is an expensive procedure. Efficient use of the cache is required to minimize the execution time, which will be bounded by memory accesses. The easiest way to ensure this is to group all accesses to one sample/pixel of data (within a solver iteration) together. Jacobian and Hessian matrices will contain many common subexpressions, therefore optimization on "the inner sum" is crucial. Since Hessian matrices are symmetric, we only need to calculate the upper triangular portion.

Given a family of Newton solvers (indexed by the power set of the set of model parameters), we can use heuristics or benchmarking to assemble them into a non-linear solver with good convergence properties.

2.3. **Recurrence Relations.** In many time-series models a simple time evolution exists. This allows the use of recurrence relations instead of explicit calculations of the model function. This greatly increases the efficiency of objective function evaluations, as well as the calculation of the Jacobian and Hessian on each solver iteration. For instance, in the case of an exponentially damped oscillatory signal, $ae^{-(d+if)t}$ of frequency f, amplitude a, and damping coefficient d, the sequence $a, ae^{-(d+if)}, ae^{-2(d+if)}, \ldots$ can be calculated using the recursion

$$z_0 = a, \quad z_{i+1} = kz_i, \quad k = e^{-(d+if)}. \tag{2.2}$$

This requires two real variables for each constant k, two variables for the most recent value z_i, and one temporary variable to do the multiplication. If $d > 0$, the sequence converges to 0 (geometrically in norm). In this case, the numerical errors do not accumulate appreciably, in the absolute sense, although they will accumulate relatively. Since errors in the measurements are commonly assumed to be uniformly and independently distributed, we are usually only concerned with absolute errors. When f is vector-valued, the same ideas often work component-wise.

2.4. **Differential Equations.**

If the model happens to have a simple dependence on the parameters, then it is possible that the derivatives that appear in the Jacobian and Hessian of 2.1 are simply expressible in terms of the model itself. Here we only illustrate what happens for a first-order dependence, which can be used to simplify both the Jacobian and the Hessian. If there is just a second-order dependence, then that can be used to simplify the Hessian.

Considering a simple model with first-order dependence on a parameter b,

$$f(b) = ae^{bp(x)} \tag{2.3}$$

then the derivative can be re-expressed in terms of f as follows.

$$\frac{\partial f}{\partial b} = p(x)ae^{bp(x)} = p(x)f(b) \tag{2.4}$$

The above can be deduced by constructing the ODE that $f(b)$ satisfies, namely

$$\frac{\partial f}{\partial b} - p(x)f(b) = 0. \tag{2.5}$$

If the dependence is algebraic - which can be considered to be a zeroth order differential equation - this can naturally be used as well for simplifications. As such dependencies are sources of redundant computations in the resulting code, it is important to factor them out.

3. Model Manipulation

The process of going from a formal mathematical model of the underlying problem, to actual C code is easiest done by first manipulating the *problem description* directly into a form more suitable for algorithmic solution. Of course, to be able to perform computer-based manipulations of a mathematical problem, we need an explicit representation of the problem that can be embedded (semantically) into software that is well suited to this task. This is where Maple shines, especially as our problem is one which involves a lot of manipulations of analytic functions.

Abstractly, what we really want to do is to be able to solve any parameter-fitting problem such as 2.1 by describing the class of functions f that we want to use, which parameters to optimize for, and how many superpositions of the basis function should be used for fitting. We can then symbolically obtain

(1) The recurrence equation satisfied by the model f with respect to the main variable t.
(2) The differential equation(s) satisfied by the model f with respect to the parameters x_i.
(3) The Jacobian of the fitting equation 2.1 with respect to all the parameters x_i.
(4) The (upper triangle of the) Hessian of the fitting equation 2.1 with respect to all of the parameters x_i.
(5) Using the recurrence and the differential equations, a reduction of the Jacobians and the Hessians with respect to the structure of the model f and the linearity of 2.1.

The first step above is obtained via `RationalNormalForms[IsHypergeometricTerm]`, a function which uses some advanced symbolic techniques to decide if a given term $f(t)$ is such that $\frac{f(t+1)}{f(t)}$ is a rational function of t, and returns this rational function if it is the case. This allows us to handle any model family f which is a *Hypergeometric term* in t. This includes functions such as $\Gamma(a * t + b)$ (and thus factorial), the pochhammer symbol $(a)_t$ (or rising factorial $a^{\bar{t}}$), the falling factorial $a^{\underline{t}}$, as well as polynomials, rational functions, and linear exponentials e^{at+b}, as well as finite products and ratios of any of the aforementioned. Our method easily generalizes to higher order recurrences, but we are not aware of a simple way to obtain these recurrences using the current version of Maple.

The second step is obtained via `gfun[holexprtodiffeq]`. The abbreviations stand respectively for *generating function* and *holonomic expression to differential equation*. The package `gfun` is described in [6] while the theory of holonomic (or D-finite) functions is described in [4]. For the time being, we can only take advantage of either zero-th and first-order differential dependence on parameters. In other words, for arbitrary functions f, g and h, we can handle models that look like $g(a) \times h(t) + f(b)$ as well as $e^{h(a)t+g(b)}$ for parameters a, b.

The Jacobians and Hessians can easily be computed via symbolic differentiation of the corresponding expressions. Note that although we could, in principle, use automatic differentiation to compute derivatives of model function code, this would obscure the role of the recurrence relations and differential equations, and make it much harder to leverage them in the reduction of code complexity.

4. Code Generation

Turning the mathematics of the previous sections into (pseudo) code, we are looking to generate something like

```
procedure GeneratedCode(y, n)
integer n, t
real f_1, f_2, ..., f_k, h_1, h_2, ..., h_k, F
real array y, Jacobian, Hessian
```

```
begin
  f_1 := f_1(0);
  h_1 := recurrence ratio of f_1;
  ...
  f_k := f_k(0);
  h_k := recurrence ratio of f_k;
  F := 0;
  Jacobian := 0;
  Hessian := 0;
  for t := 0 to (n-1) begin
    F := F + (y[t] - sum(f_i, 1 to k))^2;
    Jacobian := Jacobian + Jacobian at t;
    Jacobian := Hessian + Hessian at t;
    f_1 := h_1 * f_1;
    ...
    f_k := h_k * f_k;
  end;
return F, Jacobian, Hessian;
end;
```

In other words, we need to generate a procedure which computes F, its Jacobian and Hessian, taking full advantage of the fact that F is a sum, and that all its sub-terms satisfy a recurrence.

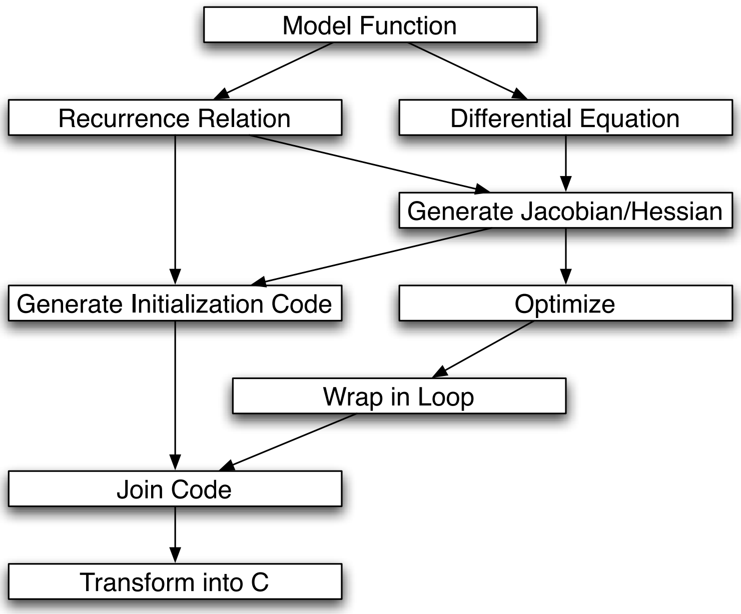
The algorithm can be explained as

(1) get recurrence relation of f on t (via `IsHypergeometricTerm`),
(2) construct Jacobian and Hessian for the model function f in terms of f,
(3) if f is a complex function, split the above into real and imaginary parts,
(4) generate code to calculate the initial value of f, the recurrence ratio h, as well as code to calculate successive terms using h and the last calculated term; do this for each superposition of f;
(5) generate code to calculate, by summing in a loop, F, Jacobian(F), Hessian(F); use previously computed relations on derivatives of f (from step (2)), as well as re-using the recurrence for f;
(6) the above code uses local variables (in the generated code) to store the Jacobian and Hessian, to enable common-subexpression elimination (as it cannot be done on Matrix/Vector entries).
(7) generate "cleanup" code to assign locally stored Jacob(F) and Hess(F) to arrays that are "returned"
(8) wrap F, Jacob(F), Hess(F) and recurrence code in a loop on t and apply sub-expression elimination optimization
(9) "paste" code together and transform to C code

There are a few things to note about the algorithm. First, f is represented abstractly in the intermediate steps of generating the code. This is useful because we know that the information derived from f is correct generically, and the details of f would actually hinder rather than help these computations. Second, all parameters of f are indexed by the superposition to which they belong.

The user inputs the model function f, the main variable t, the parameters to optimize α, and the number of superpositions k to fit.

Figure 1. Generator Control Flow



The code determines if f is a first-order recurrence on t and, if it is, extracts the recurrence ratio h. If f is not a first-order recurrence, the code terminates and prints out an appropriate error message. Next, the Jacobian of f with respect to the parameters α is computed symbolically, using the previously computed differential relations. If the differential equation technique fails for any $a \in \alpha$, that partial derivative is computed by direct symbolic differentiation. Direct symbolic differentiation is then used on the Jacobian to get the Hessian. Any occurrence of $\frac{\partial f}{\partial a}$ in the Hessian is replaced by the appropriate Jacobian entry. Since the Jacobian is given in terms of f the Hessian will be as well.

If f is a complex (vector) function, f, h (the recurrence multiplier), and the Jacobian and Hessian of f are separated into real and imaginary parts at this point. We must eventually convert all our computations to real computations only, and this point in the algorithm is where we gain the most benefit: previous computations are simpler on the complex function, while more common sub-expressions can be pulled out from the expanded version.

For each superposition of the model function (*i.e.* a peak), code is generated to calculate the initial term of the recurrence of f, the recurrence ratio h, and successive terms. To calculate the (2-norm) error function F, and the Jacobian (and Hessian of F), the following can be used where f is assumed to take values in $\mathbb{R}^m$ and y is also in $\mathbb{R}^m$, and we used α_p as a short-hand for the parameter vector.

2-norm Error Function:

$$F = \sum_{t=0}^{n-1} \left\| y(t) - \sum_{p=1}^{k} f(t, \alpha_p) \right\|^2 = \sum_{t=0}^{n-1} \sum_{s=0}^{m-1} \left(y_s(t) - \sum_{p=1}^{k} f_s(t, \alpha_p) \right)^2 \tag{4.1}$$

Jacobian:

$$J_{\alpha_q^i} = 2 \sum_{t=0}^{n-1} \sum_{s=0}^{m-1} \left(\sum_{p=1}^{k} f_s(t, \alpha_p) - y_s(t) \right) \frac{\partial f_s(t, \alpha_q)}{\partial \alpha_q^i} \tag{4.2}$$

Formulas for the Hessian can be similarly derived. It is very important to note that the formulas for the Jacobian above are completely uniform in the parameter i. This means that instead of computing each component as a sum over very similar entries, it is more efficient to compute the Jacobian as a sum of vectors, as this allows significantly more common computations to be extracted.

For every occurrence of $\frac{\partial f}{\partial a}$ in Jacob(F) and Hess(F), the corresponding entries of the pre-computed Jacob(f) and Hess(f) are substituted in. Along with code to calculate F, Jacob(F) and Hess(F), code is generated to initialize the local variables of F, Jacob(F) and Hess(F) and assign the local variables of Jacob(F) and Hess(F) to 1-D arrays.

At this point, all abstract representations of the function f are replaced by the previously computed term of f. Also, all of the indexed parameters of f in the generated code are combined into a single 1-D array. This is done to simplify use of the generated function. It provides a common interface and, if you wish to optimize two or more parameters separately, it eliminates the need for array manipulation between optimizations.

The code to calculate F, J and H is combined with the code to calculate successive terms of f. This then makes up the body of a loop on the main variable t. Common sub-expresion elimination is used on the loop body via `codegen[optimize]` with the `tryhard` option, and the optimized code is wrapped in a loop on t from 0 to $n-1$, where n (number of data points) is an argument of the generated function. The loop is then spliced with the previous code, transformed into a C function with arguments `(double *y, int n, double *gama, double *J, double *H)` and the return value F. Finally, the C code is output to a file specified by the user.

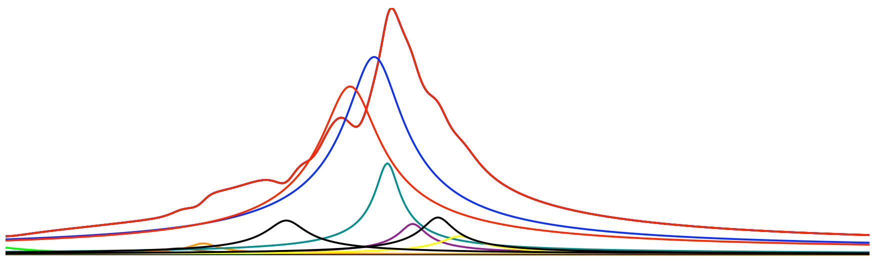


FIGURE 2. Soya bean oil phantom 1H-MR spectrum (maroon) and component estimates.

5. Applications

We give two examples from Magnetic Resonance. The first example provided the impetus for this work, and in addition to developing the model functions, we sketch the arguments for using a series of subspace searches to try to find the global minimum for a highly non-convex function. The second shows the use of real exponentials.

5.1. Magnetic Resonance Spectroscopy.

MRS provides a method for quantifying the concentrations of chemical constituents of a given tissue sample, both in vitro and in vivo.

Signal in conventional MRS is attained from hydrogen atoms bound to the molecules of interest. The different bonding patterns found in these molecules slightly alter the base resonance frequency of the hydrogens and lead to an effect referred to as chemical shift. Other factors such as bulk magnetic susceptibility and tissue orientation (with respect to the magnetic field) also contribute to chemical shift, but generally to a lesser extent. Different chemicals can therefore be identified by their frequency shift relative to some reference (usually water). In order to quantify the amounts of chemical present, we model the signal generation and find the maximum likelihood estimate for the model parameters. Figure 2 shows the resulting decomposition of a measured spectrum.

Unfortunately this objective function is not convex. By approximating the problem by one which is convex, and then successively introducing greater complexity in successive approximations, and switching between the frequency and time domains, we are able to stay within the basin of convergence of Newton's method for the successive problems. We can ignore the signal phase while still fitting the peak positions in the frequency domain, and then switch to the time domain, increasing the accuracy of the determined model parameters. Using prior knowledge of fixed peak resonance frequency relationships of the compounds we expect to find in the samples, we both increase the accuracy of our estimates, and–because we are using symbolic code generation–reduce the complexity and memory footprint of the solver. We find that this method fits spectra very quickly and provides good results. To understand the source of non-convexity and the need for multiple solution stages, we briefly outline the models in this section.

The time-domain signal that is measured is the free induction decay (FID), which is the superposition of signals from several different tissues, each having the form

$$ae^{-(d+if)t} \tag{5.1}$$

where $a \in \mathbb{C}$ is the signal strength and phase, d is the damping (called $1/T_2$ in the MR literature), and f is the frequency (peak position).

The Fourier Transform of one signal is a Lorentzian:

$$\frac{1}{d+i(f-x)} = \frac{d}{d^2+(f-x)^2} + i\frac{x-f}{d^2+(f-x)^2}.$$

A delay in sampling will create a complex phase ramp e^{ikt} where k is related to the delay. Since spectroscopy pulse sequences are complicated, including water suppression, the phases of the different resonances are unlikely to be exactly the same.

The squared norm of the spectrum is not affected by delays in sampling, nor by the overall phase of the signal. If peaks are well-separated, or have similar phase, the spectrum is not affected by the phase differences between different peaks either. Under these assumptions, we can fit the peaks without worrying about phase. This reduces the dimensionality of the problem and eliminates a large source of non-convexity caused by the ambiguity in phase angles.

Fitting a single peak is equivalent to minimizing the convex objective:

$$\min_{a,d,f} \sum_x \left(\mathcal{F}y(x) - \frac{a\bar{a}}{d^2+(f-x)^2} \right)^2 .$$

Once we add a series of peaks, we must take the norm of the sum of the real and imaginary parts, and not the sum of the norms. This makes the model function more complicated, and again breaks the convexity of the objective function. For example, if we have two well-separated peaks, the objective function will have a global minimum at the correct fit and also a local minimum where the left model peak is fitting the right observed peak or vice versa.

To do unattended peak fitting we approximate the problem with a (simplified) convex problem; its solution can then be used as a good starting point for Newton's method to converge to the global minimum of the non-convex objective function. This is achieved by smoothing out the peaks until the spectrum itself becomes convex. We add in a damping factor k and multiply the time-domain data by e^{-kt}, and then try to fit it with a sum of Lorentzians with d replaced by kd. As the Newton method converges, the damping is iteratively decreased until the original problem is recovered.

Writing the objective function in terms of the FID or its DFT is almost equivalent, since adding white noise before a DFT is equivalent to adding it afterwards. We have seen that fitting the spectrum in the frequency domain has the advantage that we can easily fit individual peaks in the magnitude as a way of obtaining an initial guess. Time-domain fitting,

$$\min_{a_0,d_0,f_0,\ldots,d_k,f_k} \sum_t \left\| y(t) - \sum_{s\in\{\text{sources}\}} a_p e^{-(d_p+if_p)t} \right\|^2, \tag{5.2}$$

(where $y(t)$ is the complex sample at time t), has the advantage that there is no Gibb's ringing, it is easier to fit asymmetric echos, and the model functions can be calculated using only additions and multiplications via the recurrence relations given in equation (2.2).

In the frequency domain, the magnitude of the peaks is the easiest and most intuitive to fit. In the model we use for our specific application, there are theoretically 12 distinct signals that compose the FID. Some of these peaks will always have fixed frequency offsets from one another, and generally all 12 peaks should appear in more or less fixed locations. This fact allows us to greatly simplify the process of finding an initial guess by first fitting the location of the set of fixed peaks. In a second step, we can individually fit the frequencies in the time domain. Fitting the complex data in the time domain is most useful for determining the peak area, as the complex area values carry the signal phase information with them. This becomes a linear problem with a quadratic objective function, yielding a best fit solution in only one iteration of the solver. Performing this operation in the frequency domain would require guessing the phase angles. In addition to fitting the area, attaining accurate values for damping is also simplified in the time domain. In our method, the complex data is iteratively fit for area, phase and damping, until the change in residual area after each iteration is very small.

Although all of the inner loops were generated symbolically, we only benefit from the improvements in this paper when fitting in the time domain.

5.2. **Magnetic Resonance Relaxometry.** A real-valued example problem can be drawn from MR Relaxometry, where the purpose of experiment is not to acquire an image or FID spectrum as in MRI or MRS, but to determine the time constants of the signal decay, which vary due to chemical environment, including pH levels and temperature, making relaxometry a useful non-invasive tool for determination of these quantities *in-vivo* in real time. Changes in time constants are also useful as indicators of disease not apparent when examining MR images, such as susceptibility for seizure [5].

The time constants arise from the Bloch equations [3], which govern the response of a proton in a magnetic field. The decay constant $T1$ relates to the longitudinal relaxation (spin-lattice interactions) of the protons, and manifests itself in the re-growth of the proton magnetization in the direction of the main field. The decay constant $T2$, or transverse relaxation (spin-spin interaction) is a measure of the rate of signal de-phasing. Depending on the set-up of the MR Relaxometry experiment, one can measure either constant.

The typical MR Relaxometry experiment involves exciting the sample as usual, and waiting for some time before measuring the magnitude of the MR signal. This is repeated with a small number (typically 6-10) of different measurement delays, which have the form of a real-valued, damped exponential,

$$ae^{-dt}. \tag{5.3}$$

When information is required about the decay constants of multiple species present in one sample, the problem becomes very similar to that of spectroscopy, where the objective function has the form:

$$\min_{a_0,d_0,\ldots,a_k,d_k} \sum_t \left\| y(t) - \sum_{s\in\{\text{sources}\}} a_p e^{-d_p t} \right\|^2 . \tag{5.4}$$

In order to make use of MR Relaxometry for real-time applications, quick and accurate estimation of d is necessary.

Symbolic code generation made the development of this multitude of fast solvers easy.

6. Results

Performance testing of the generated code for calculation of the Hessian and Jacobian was performed for both a real and a complex model function (equations (5.4) and (5.2)) with 12 signal sources. The execution time was measured as the average of 1000 iterations of the solver on a 1.33 GHz PowerPC G4 processor, over 1024 pseudo-random sample data points. All C code was compiled with GCC 3.3, with full optimizations (-O3) enabled.

The benchmarks were run with different combinations of Maple-based optimizations enabled: no Maple-level optimizations, incorporation of the recurrence relation (R), incorporation of the symbolic differential equations (D), and both (R + D). In addition, different common sub-expression elimination routines in Maple were tested, as detailed in the tables.

Tables 1 and 2 give the run time and relative speed improvements (with respect to a non-optimized version) of the generated code for the real-valued model function, where the derivatives are taken with respect to the d_i terms in the exponential.

Tables 3 and 4 present the data for the complex-valued objective, with derivatives that are taken with respect to the imaginary f_i terms in the exponential.

Lastly, tables 5 and 6 present the data for the complex-valued objective, but with derivatives that are taken with respect to the real d_i terms in the exponential. As one would expect (since both f and d are in the exponent), the performance of generating Hessians and Jacobians for d is nearly identical to the cost of doing so for f.

The trend is clear: in all cases, a large benefit is to be gained by exploiting the problem structure. Not surprisingly, Maple's common sub-expression elimination by itself yields a large improvement in execution time, as the 'base' code contains sums of constant exponential terms (as well as sines and cosines in the complex case). While the recurrence relation and differential equation optimizations alone lend a small speed increase, the maximum speedup is gained when both are put to work. This jump in performance is due to the fact that when working together, all exponential and trigonometric function calls are eliminated from the loop over the data points, (see section 4 for pseudo-code) leaving only addition and multiplication operations.

Table 1. Time for generation of one Hessian and Jacobian for real model function (5.4) with 12 sources and 1024 sample points. Derivatives taken w.r.t. d.

	Time per iteration (seconds).			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	0.2340	0.0622	0.2283	0.0060
codegen[optimize]	0.0111	0.0077	0.0103	0.0033
'tryhard' optimize	0.0079	0.0059	0.0080	0.0019

Table 2. Average speed improvement factor for real model function (5.4) with 12 sources and 1024 sample points. Derivatives taken w.r.t. d.

	Relative speedup due to optimizations.			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	1	4	1	39
codegen[optimize]	21	30	23	72
'tryhard' optimize	30	40	29	124

7. Conclusion

Our results are very promising. We are continuing to improve the underlying infrastructure, to be able to deal with wider classes of models. In particular, we should be able to use models which satisfy a linear recurrence equation of any (fixed, small) order. We also hope to be able to generalize to multi-dimensional recurrences, for example for both scalar and vector data on a 2 dimensional grid; naturally higher dimensions are possible as well.

We also want to further experiment and see more precisely what the source of the common sub-expressions that remain is. We have quite conclusively

Table 3. Time for generation of one Hessian and Jacobian for complex model function (5.2) with 12 sources and 1024 sample points. Derivatives taken w.r.t. f.

	Time per iteration (seconds).			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	1.1522	0.4121	0.5829	0.0038
codegen[optimize]	0.0245	0.0208	0.0202	0.0021
'tryhard' optimize	0.0226	0.0196	0.0194	0.0021

Table 4. Average speed improvement factor for complex model function (5.2) with 12 sources and 1024 sample points. Derivatives taken w.r.t. f.

	Relative speedup due to optimizations.			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	1	3	2	300
codegen[optimize]	47	55	57	548
'tryhard' optimize	51	59	59	547

Table 5. Time for generation of one Hessian and Jacobian for complex model function (5.2) with 12 sources and 1024 sample points. Derivatives taken w.r.t. d.

	Time per iteration (seconds).			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	1.1237	0.4037	0.5712	0.0037
codegen[optimize]	0.0231	0.0196	0.0198	0.0021
'tryhard' optimize	0.0227	0.0197	0.0191	0.0021

Table 6. Average speed improvement factor for complex model function (5.2) with 12 sources and 1024 sample points. Derivatives taken w.r.t. d.

	Relative speedup due to optimizations.			
	Base	Base + R	Base + D	Base + R + D
No Maple optim.	1	3	2	301
codegen[optimize]	49	57	57	541
'tryhard' optimize	50	57	59	546

shown that taking advantage of the structure of the problem (linearity because of the summation, recurrence equations, differential equations, symmetry of the Hessian) leads to very significant speedups. We would like to understand (as much as possible) what is the underlying structure which accounts for the many common sub-expressions left and which, when eliminated, contribute such an additional speedup factor.

It is worthwhile remembering that these calculations would either be required for all pixels in an image (millions of solves) for relaxometry, or for optimization across all patients in a clinical study for spectroscopy, which means several gigabytes of data processed in hundreds of iterations. Therefore speedups such as the ones we demonstrate are not merely academic, but make a substantial impact on the size of problems that can be tackled. As these are important health-related applications, we believe this is a very promising area of research.

Finally, it is worth noting that while we have concentrated on spectroscopy and relaxometry applications in this paper, we are aware of quite a number of applications which are mathematically quite similar, but from widely different domains. We still need to ascertain if, in practice, we see gains which are as significant as the current application. We certainly are hopeful that this will be the case.

References

[1] C. Anand, J. Carette, and A. Korobkine. Visual tracking employing Maple code generation. In *Maple Summer Workshop 2004*, 2004.

[2] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation*, pages 242–249, 1994.

[3] F. Bloch. Nuclear induction. *Phys. Rev.*, 70:460, 1946.

[4] Frédéric Chyzak and Bruno Salvy. Non-commutative elimination in Ore algebras proves multivariate holonomic identities. *Journal of Symbolic Computation*, 26(2):187–227, August 1998.

[5] Andres M. Kanner. Abnormalities identified with t2 relaxometry in hippocampi remote from the seizure focus: Do they mean anything?. *Epilepsy Currents*, 4(3):120–121, 2004.

[6] Bruno Salvy and Paul Zimmermann. Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

McMaster University, Department of Computing and Software
E-mail address: {anandc,carette,curtia,milled3}@mcmaster.ca